下载APP

02 | RDD与编程模型：延迟计算是怎么回事？

2021-09-13 吴磊

《零基础入门Spark》 课程介绍 >

讲述：吴磊

时长 19:05 大小 17.48M

你好，我是吴磊。

在上一讲，我们一起开发了一个 Word Count 小应用，并把它敲入到 spark-shell 中去执行。Word Count 的计算步骤非常简单，首先是读取数据源，然后是做分词，最后做分组计数、并把词频最高的几个词汇打印到屏幕上。

如果你也动手实践了这个示例，你可能会发现，在 spark-shell 的 REPL 里，所有代码都是立即返回、瞬间就执行完毕了，相比之下，只有最后一行代码，花了好长时间才在屏幕上打印出 the、Spark、a、and 和 of 这几个单词。

针对这个现象，你可能会觉得很奇怪：“读取数据源、分组计数应该是最耗时的步骤，为什么它们瞬间就返回了呢？打印单词应该是瞬间的事，为什么这一步反而是最耗时的

呢？”要解答这个疑惑，我们还是得从 RDD 说起。

什么是 RDD

为什么非要从 RDD 说起呢？首先，RDD 是构建 Spark 分布式内存计算引擎的基石，很多 Spark 核心概念与核心组件，如 DAG 和调度系统都衍生自 RDD。因此，深入理解 RDD 有利于你更全面、系统地学习 Spark 的工作原理。

其次，尽管 RDD API 使用频率越来越低，绝大多数人也都已经习惯于 DataFrame 和 Dataset API，但是，无论采用哪种 API 或是哪种开发语言，你的应用在 Spark 内部最终都会转化为 RDD 之上的分布式计算。换句话说，如果你想要对 Spark 作业有更好的把握，前提是你要对 RDD 足够了解。

既然 RDD 如此重要，那么它到底是什么呢？用一句话来概括，**RDD 是一种抽象，是 Spark 对于分布式数据集的抽象，它用于囊括所有内存中和磁盘中的分布式数据实体。**

在上一讲中，我们把 RDD 看作是数组，咱们不妨延续这个思路，通过对比 RDD 与数组之间的差异认识一下 RDD。

我列了一个表，做了一下 RDD 和数组对比，你可以先扫一眼：

对比项	数组	RDD
概念类型	数据结构实体	数据模型抽象
数据跨度	单机进程内	跨进程、跨计算节点
数据构成	数组元素	数据分片（Partitions）
数据定位	数组下标、索引	数据分片索引

极客时间

RDD与数组的对比

我在表中从四个方面对数组和 RDD 进行了对比，现在我来详细解释一下。

首先，就概念本身来说，数组是实体，它是一种存储同类元素的数据结构，而 RDD 是一种抽象，它所囊括的是分布式计算环境中的分布式数据集。

因此，这两者第二方面的不同就是在活动范围，数组的“活动范围”很窄，仅限于单个计算节点的某个进程内，而 RDD 代表的数据集是跨进程、跨节点的，它的“活动范围”是整个集群。

至于数组和 RDD 的第三个不同，则是在数据定位方面。在数组中，承载数据的基本单元是元素，而 RDD 中承载数据的基本单元是数据分片。在分布式计算环境中，一份完整的数据集，会按照某种规则切割成多份数据分片。这些数据分片被均匀地分发给集群内不同的计算节点和执行进程，从而实现分布式并行计算。

通过以上对比，不难发现，**数据分片**（Partitions）是 RDD 抽象的重要属性之一。在初步认识了 RDD 之后，接下来咱们换个视角，从 RDD 的重要属性出发，去进一步深入理解 RDD。要想吃透 RDD，我们必须掌握它的 4 大属性：

partitions：数据分片

partitioner：分片切割规则

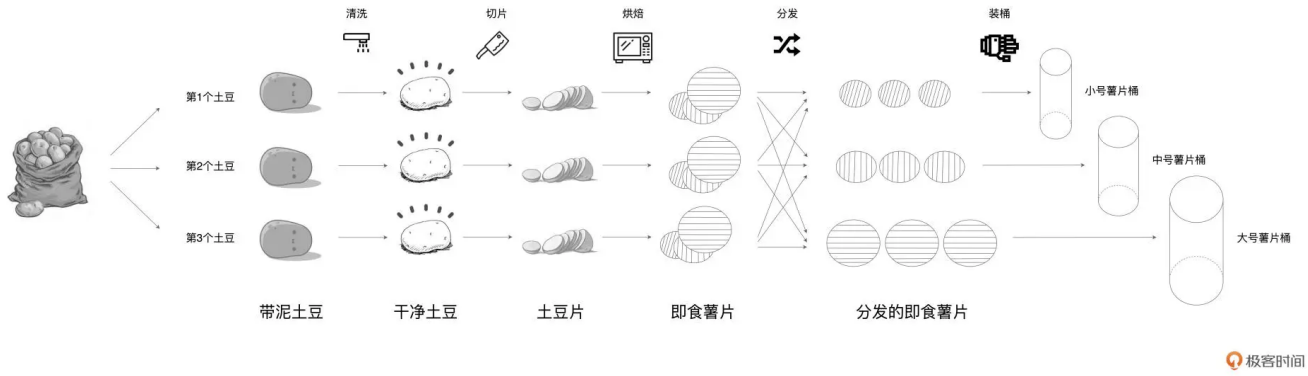
dependencies：RDD 依赖

compute：转换函数

如果单从理论出发、照本宣科地去讲这 4 大属性，未免过于枯燥、乏味、没意思！所以，我们从一个制作薯片的故事开始，去更好地理解 RDD 的 4 大属性。

从薯片的加工流程看 RDD 的 4 大属性

在很久很久以前，有个生产桶装薯片的工坊，工坊的规模较小，工艺也比较原始。为了充分利用每一颗土豆、降低生产成本，工坊使用 3 条流水线来同时生产 3 种不同尺寸的桶装薯片。3 条流水线可以同时加工 3 颗土豆，每条流水线的作业流程都是一样的，分别是清洗、切片、烘焙、分发和装桶。其中，分发环节用于区分小、中、大号 3 种薯片，3 种不同尺寸的薯片分别被发往第 1、2、3 条流水线。具体流程如下图所示。



RDD的生活化类比

好了，故事讲完了。那如果我们把每一条流水线看作是分布式运行环境的计算节点，用薯片生产的流程去类比 Spark 分布式计算，会有哪些有趣的发现呢？

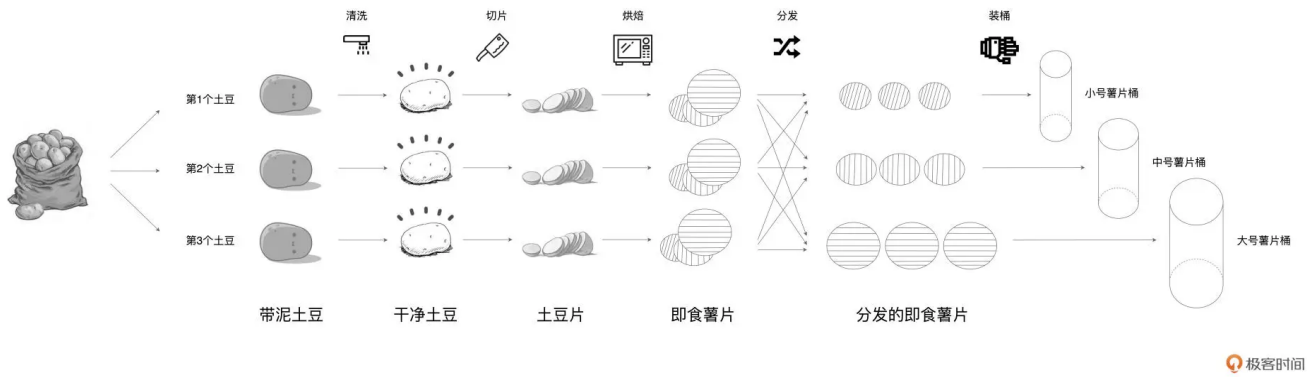
显然，这里的每一种食材形态，如“带泥土豆”、“干净土豆”、“土豆片”等，都可以看成是一个个 RDD。而**薯片的制作过程，实际上就是不同食材形态的转换过程**。

起初，工人们从麻袋中把“带泥土豆”加载到流水线，这些土豆经过清洗之后，摇身一变，成了“干净土豆”。接下来，流水线上的切片机再把“干净土豆”切成“土豆片”，然后紧接着把这些土豆片放进烤箱。最终，土豆片烤熟之后，就变成了可以放心食用的即食薯片。

通过分析我们不难发现，不同食材形态之间的转换过程，与 Word Count 中不同 RDD 之间的转换过程如出一辙。

所以接下来，我们就结合薯片的制作流程，去理解 RDD 的 4 大属性。

首先，咱们沿着纵向，也就是从上到下的方向，去观察上图中土豆工坊的制作工艺。



RDD的生活化类比

我们可以看到对于每一种食材形态来说，流水线上都有多个实物与之对应，比如，“带泥土豆”是一种食材形态，流水线上总共有 3 颗“脏兮兮”的土豆同属于这一形态。

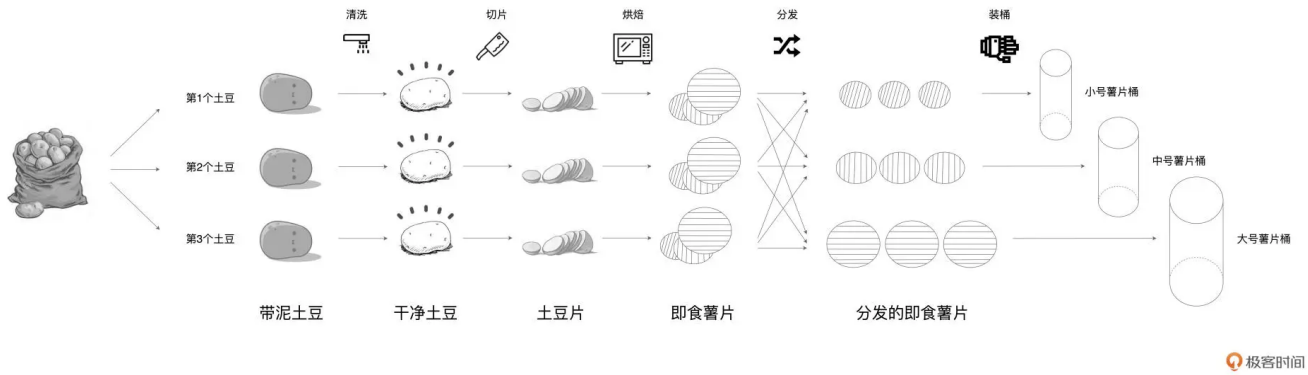
如果把“带泥土豆”看成是 RDD 的话，那么 RDD 的 partitions 属性，囊括的正是麻袋里那一颗颗脏兮兮的土豆。同理，流水线上所有洗净的土豆，一同构成了“干净土豆”RDD 的 partitions 属性。

我们再来看 RDD 的 partitioner 属性，这个属性定义了把原始数据集切割成数据分片的切割规则。在土豆工坊的例子中，“带泥土豆”RDD 的切割规则是随机拿取，也就是从麻袋中随机拿取一颗脏兮兮的土豆放到流水线上。后面的食材形态，如“干净土豆”、“土豆片”和“即食薯片”，则沿用了“带泥土豆”RDD 的切割规则。换句话说，后续的这些 RDD，分别继承了前一个 RDD 的 partitioner 属性。

这里面与众不同的是“分发的即食薯片”。显然，“分发的即食薯片”是通过对“即食薯片”按照大、中、小号做分发得到的。也就是说，对于“分发的即食薯片”来说，它的 partitioner 属性，重新定义了这个 RDD 数据分片的切割规则，也就是把先前 RDD 的数据分片打散，按照薯片尺寸重新构建数据分片。

由这个例子我们可以看出，数据分片的分布，是由 RDD 的 partitioner 决定的。因此，RDD 的 partitions 属性，与它的 partitioner 属性是强相关的。

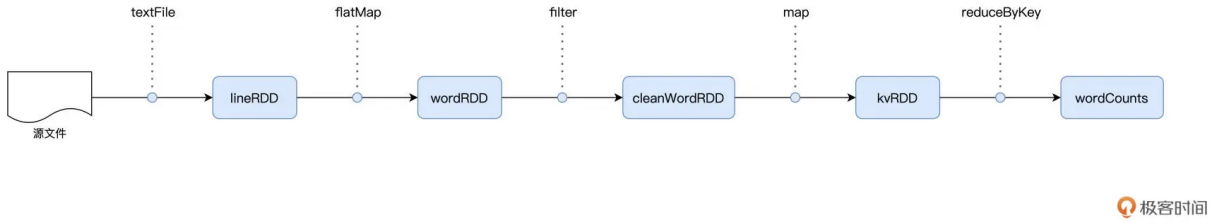
横看成岭侧成峰，很多事情换个视角看，相貌可能会完全不同。所以接下来，我们横向地，也就是沿着从左至右的方向，再来观察土豆工坊的制作工艺。



RDD的生活化类比

不难发现，流水线上的每一种食材形态，都是上一种食材形态在某种操作下进行转换得到的。比如，“土豆片”依赖的食材形态是“干净土豆”，这中间用于转换的操作是“切

片”这个动作。回顾 Word Count 当中 RDD 之间的转换关系，我们也会发现类似的现象。



Word Count中的RDD转换

在数据形态的转换过程中，每个 RDD 都会通过 dependencies 属性来记录它所依赖的前一个、或是多个 RDD，简称“父 RDD”。与此同时，RDD 使用 compute 属性，来记录从父 RDD 到当前 RDD 的转换操作。

拿 Word Count 当中的 wordRDD 来举例，它的父 RDD 是 lineRDD，因此，它的 dependencies 属性记录的是 lineRDD。从 lineRDD 到 wordRDD 的转换，其所依赖的操作是 flatMap，因此，wordRDD 的 compute 属性，记录的是 flatMap 这个转换函数。

总结下来，薯片的加工流程，与 RDD 的概念和 4 大属性是一一对应的：

不同的食材形态，如带泥土豆、土豆片、即食薯片等等，对应的就是 RDD 概念；

同一种食材形态在不同流水线上的具体实物，就是 RDD 的 partitions 属性；

食材按照什么规则被分配到哪条流水线，对应的就是 RDD 的 partitioner 属性；

每一种食材形态都会依赖上一种形态，这种依赖关系对应的是 RDD 中的 dependencies 属性；

不同环节的加工方法对应 RDD 的 compute 属性。

在你理解了 RDD 的 4 大属性之后，还需要进一步了解 RDD 的编程模型和延迟计算。编程模型指导我们如何进行代码实现，而延迟计算是 Spark 分布式运行机制的基础。只有搞明白编程模型与延迟计算，你才能流畅地在 Spark 之上做应用开发，在实现业务逻辑的同时，避免埋下性能隐患。

编程模型与延迟计算

你还记得我在上一讲的最后，给你留的一道思考题吗：map、filter、flatMap 和 reduceByKey 这些算子，有哪些共同点？现在我们来揭晓答案：

首先，这 4 个算子都是作用（Apply）在 RDD 之上、用来做 RDD 之间的转换。比如，flatMap 作用在 lineRDD 之上，把 lineRDD 转换为 wordRDD。

其次，这些算子本身是函数，而且它们的参数也是函数。参数是函数、或者返回值是函数的函数，我们把这类函数统称为“**高阶函数**”（Higher-order Functions）。换句话说，这 4 个算子，都是高阶函数。

关于高阶函数的作用与优劣势，我们留到后面再去展开。这里，我们先专注在 RDD 算子的第一个共性：**RDD 转换**。

RDD 是 Spark 对于分布式数据集的抽象，**每一个 RDD 都代表着一种分布式数据形态**。比如 lineRDD，它表示数据在集群中以行（Line）的形式存在；而 wordRDD 则意味着数据的形态是单词，分布在计算集群中。

理解了 RDD，那什么是 RDD 转换呢？别着急，我来以上次 Word Count 的实现代码为例，来给你讲一下。以下是我们上次用的代码：

复制代码

```
import org.apache.spark.rdd.RDD
val rootPath: String = _
val file: String = s"${rootPath}/wikiOfSpark.txt"
// 读取文件内容
val lineRDD: RDD[String] = spark.sparkContext.textFile(file)
// 以行为单位做分词
val wordRDD: RDD[String] = lineRDD.flatMap(line => line.split(" "))
val cleanWordRDD: RDD[String] = wordRDD.filter(word => !word.equals(""))
// 把RDD元素转换为（Key，Value）的形式
val kvRDD: RDD[(String, Int)] = cleanWordRDD.map(word => (word, 1))
// 按照单词做分组计数
val wordCounts: RDD[(String, Int)] = kvRDD.reduceByKey((x, y) => x + y)
// 打印词频最高的5个词汇
wordCounts.map{case (k, v) => (v, k)}.sortByKey(false).take(5)
```

回顾 Word Count 示例，我们会发现，Word Count 的实现过程，实际上就是不同 RDD 之间的一个转换过程。仔细观察我们会发现，Word Count 示例中一共有 4 次 RDD 的转换，我来具体解释一下：

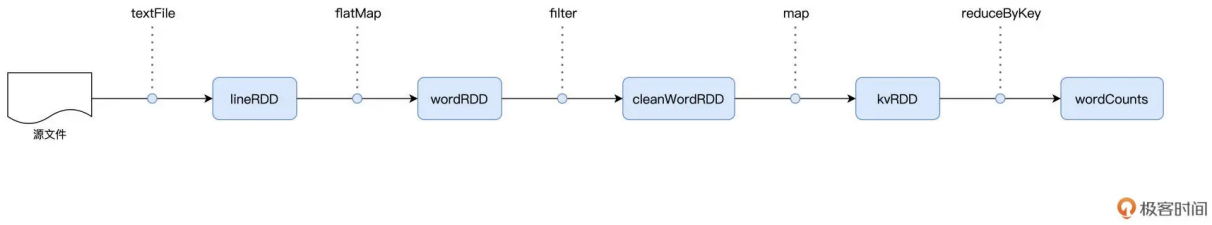
起初，我们通过调用 textFile API 生成 lineRDD，然后用 flatMap 算子把 lineRDD 转换为 wordRDD；

接下来，filter 算子对 wordRDD 做过滤，并把它转换为不带空串的 cleanWordRDD；

然后，为了后续的聚合计算，map 算子把 cleanWordRDD 又转换成元素为（Key，Value）对的 kvRDD；

最终，我们调用 reduceByKey 做分组聚合，把 kvRDD 中的 Value 从 1 转换为单词计数。

这 4 步转换的过程如下图所示：



Word Count中的RDD转换

我们刚刚说过，RDD 代表的是分布式数据形态，因此，**RDD 到 RDD 之间的转换，本质上是数据形态上的转换（Transformations）**。

在 RDD 的编程模型中，一共有两种算子，Transformations 类算子和 Actions 类算子。开发者需要使用 Transformations 类算子，定义并描述数据形态的转换过程，然后调用 Actions 类算子，将计算结果收集起来、或是物化到磁盘。

在这样的编程模型下，Spark 在运行时的计算被划分为两个环节。

1. 基于不用数据形态之间的转换，构建**计算流图**（DAG，Directed Acyclic Graph）；

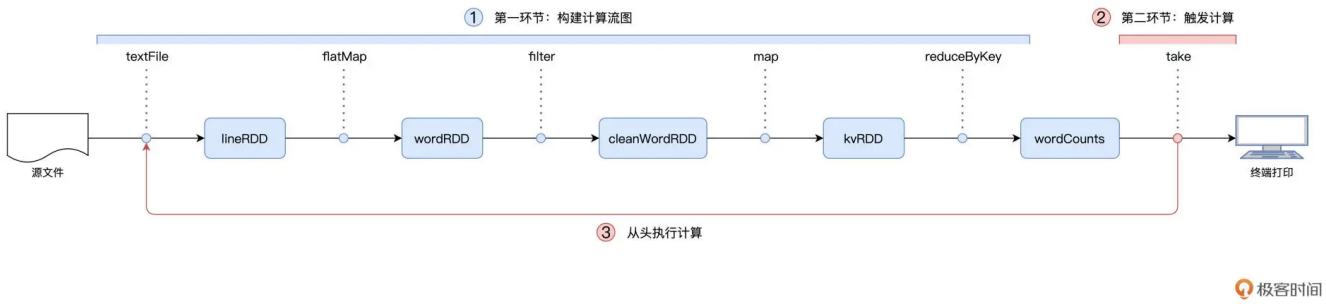
2. 通过 Actions 类算子，以**回溯的方式去触发执行**这个计算流图。

换句话说，开发者调用的各类 Transformations 算子，并不立即执行计算，当且仅当开发者调用 Actions 算子时，之前调用的转换算子才会付诸执行。在业内，这样的计算模式有个专门的术语，叫作“**延迟计算**”（Lazy Evaluation）。

延迟计算很好地解释了本讲开头的问题：为什么 Word Count 在执行的过程中，只有最后一行代码会花费很长时间，而前面的代码都是瞬间执行完毕的呢？

这里的答案正是 Spark 的延迟计算。flatMap、filter、map 这些算子，仅用于构建计算流图，因此，当你在 spark-shell 中敲入这些代码时，spark-shell 会立即返回。只有在你敲入最后那行包含 take 的代码时，Spark 才会触发执行从头到尾的计算流程，所以直观地看上去，最后一行代码是最耗时的。

Spark 程序的整个运行流程如下图所示：



延迟计算

你可能会问：“在 RDD 的开发框架下，哪些算子属于 Transformations 算子，哪些算子是 Actions 算子呢？”

我们都知道，Spark 有很多算子，Spark 官网提供了完整的 RDD 算子集合，不过对于这些算子，官网更多地是采用一种罗列的方式去呈现的，没有进行分类，看得人眼花缭乱、昏昏欲睡。因此，我把常用的 RDD 算子进行了归类，并整理到了下面的表格中，供你随时查阅。

算子类型	适用范围	算子用途	算子集合		
Transformations	任意RDD	单个RDD数据转换	map mapPartitions mapPartitionsWithIndex	filter	flatMap
	任意RDD	与其他RDD进行组合	union intersection	join cogroup cartesian	
	任意RDD	数据采样	sample	distinct	
	任意RDD	数据分片的重分布	coalesce	repartition repartitionAndSortWithinPartitions	
	Paired RDD	单个RDD数据聚合	groupByKey sortByKey	reduceByKey aggregateByKey	
Actions	任意RDD	收集数据到Spark Driver	collect first take	takeSample takeOrdered	count
	任意RDD	数据持久化到存储系统	saveAsTextFile saveAsSequenceFile saveAsObjectFile		
	任意RDD	函数式编程副作用操作	foreach		

极客时间

图片整理自：https://spark.apache.org/docs/latest/rdd-programming-guide.html

结合每个算子的分类、用途和适用场景，这张表格可以帮你更快、更高效地选择合适的算子来实现业务逻辑。对于表格中不熟悉的算子，比如 aggregateByKey，你可以结合官网的介绍与解释，或是进一步查阅网上的相关资料，有的放矢地去深入理解。重要的算子，我们会在之后的课里详细解释。

重点回顾

今天这一讲，我们重点讲解了 RDD 的编程模型与延迟计算，并通过土豆工坊的类比介绍了什么是 RDD。**RDD 是 Spark 对于分布式数据集的抽象，它用于囊括所有内存中和磁盘中的分布式数据实体。**对于 RDD，你要重点掌握它的 4 大属性，这是我们后续学习的重要基础：

partitions：数据分片

partitioner：分片切割规则

dependencies：RDD 依赖

compute：转换函数

深入理解 RDD 之后，你需要熟悉 RDD 的编程模型。在 RDD 的编程模型中，开发者需要使用 Transformations 类算子，定义并描述数据形态的转换过程，然后调用 Actions 类算子，将计算结果收集起来、或是物化到磁盘。

而延迟计算指的是，开发者调用的各类 Transformations 算子，并不会立即执行计算，当且仅当开发者调用 Actions 算子时，之前调用的转换算子才会付诸执行。

每课一练

对于 Word Count 的计算流图与土豆工坊的流水线工艺，尽管看上去毫不相关，风马牛不相及，不过，你不妨花点时间想一想，它们之间有哪些区别和联系？

欢迎你把答案分享到评论区，我在评论区等你，也欢迎你把这一讲分享给更多的朋友和同事，我们下一讲见！

分享给需要的人，Ta订阅后你可得 20 元现金奖励

赞 6 提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。 页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 01 | Spark：从“大数据的Hello World”开始

下一篇 03 | RDD常用算子（一）：RDD内部的数据转换

更多学习推荐

精选留言 (3)

写留言

AIK
2021-09-13

从两者的整体流程来看，结果都是不可逆的，但是WordCount可以设置Cache和Checkpoint，方便加速访问和故障修复，而土豆加工流程却不可以。土豆加工流程是DAG的概图。

作者回复: Perfect~

7

LJK
2021-09-18

Action中漏了一个reduce，今天才注意到reduceByKey是transformation而reduce是action

作者回复: 是的，感谢提醒~

1

行走的大数据

2021-09-16

从两者不同点看，workdcount切切实实为延迟计算，而土豆工坊的流程为切实发生的。

作者回复: 很棒的视角~ 一个是延迟计算（Lazy evaluation），一个是“及早求值”（Early evaluation）

1